Pattern Matching

This material is not covered in our textbook but you can read about pattern matching in Chapter I.3 of Algorithms on Strings, Trees, and Sequences by Gusfield.

In many situations, we are searching for a type of string rather than a particular string, such as for a telephone number, an address, a gene, or what have you. In other words, we are looking for a string in a family, and this family may be large or even infinite. It is therefore not feasible to list all strings in the family and search for each one. We use regular expressions to specify such families of strings.

Regular Expressions. Consider the problem of specifying a proper real number. For simplicity, we consider only unsigned fixed-point representations in binary notation. First we need an alphabet, which in this case is \( \Sigma = \{ 0, 1, \cdot \} \). Next we need to agree on a convention:

- the sequence starts with a non-empty sequence of digits of which the first is non-zero except if no other digit follows;
- there is possibly a decimal point and, if there is, that point is followed by a non-empty sequence of digits of which the last one is non-zero.

We write these conditions more formally in set notation as \( \{ \{ 1 \} \{ 0, 1 \} \cup \{ 0 \} \}\{ \{ 1 \} \{ 0, 1 \} \}^* \{ 1 \} \}. \) Here, \( \varepsilon \) represents the empty string. For two sets \( X \) and \( Y \), \( X \cup Y \) is the union, \( XY = (X)(Y) = \{ ab \mid a \in X, b \in Y \} \) is the set of concatenations, and \( X^* = (X)^* = \bigcup_{i=0}^{\infty} X^i \) is the closure under concatenation, with \( X^0 = \{ \varepsilon \} \) and \( X^i = XX^{i-1} \) for all positive \( i \). In words, \( X^* \) is the concatenation of arbitrarily many strings from \( X \), possibly with repetitions. We use regular expressions from which similar expressions in set notation can be derived. We define them recursively assuming an alphabet \( \Sigma \) that does not contain the special symbols \( \varepsilon, (\cdot), \cup, \text{ and } ^* \):

1. \( \varepsilon \) and \( a \) for all \( a \in \Sigma \) are regular expressions;
2. if \( A \) and \( B \) are regular expressions then so are \( A \cup B, (A)(B) \) and \( (A)^* \).

The length \( m = |A| \) of \( A \) is the number of symbols we need to write \( A \), counting \( \varepsilon \), characters and operations but not parentheses. The string set \( S(A) \) is again defined recursively. For single-letter expressions, we have \( S(\varepsilon) = \{ \varepsilon \} \) and \( S(a) = \{ a \} \). Otherwise, the expression is a union, concatenation or closure of one or two smaller expressions: \( S(A \cup B) = S(A) \cup S(B) \), \( S((A)(B)) = S(A)S(B) \), and \( S((A)^*) = S(A)^* \). Given a regular expression \( A \) and a text \( T \), we consider the pattern matching problem of deciding whether or not \( T \in S(A) \). Even though this sounds like an easier problem than deciding whether a substring of \( T \) belongs to \( S(A) \), the two are really the same. Specifically, a substring of \( T \) belongs to \( S(A) \) iff \( T \) belongs to \( S(\Sigma^* A \Sigma^*) \).

Nondeterministic Finite State Machine. Similar to string matching, we use finite state machines for pattern matching. However, because the pattern may correspond to a large and possibly infinite set of strings, we can no longer follow a deterministic sequence of transitions. We need the more powerful nondeterministic finite state machine that can guess the appropriate transitions that lead to the positive answer, if any.

As before, we represent the finite state machine by a directed graph, but it is now convenient to label the edges with characters from the alphabet. We construct the digraph, \( G \), from the regular expression, \( A \). It has a unique source vertex (the initial state) and a unique sink vertex (the final state of the finite state machine). The two base cases are illustrated in Figure 111. We construct larger

![Figure 111](image-url)
digraphs by combining smaller digraphs of regular subexpressions, as shown in Figure 112.

![Diagram of digraphs](image1)

Figure 112: The three constructions correspond to the three operations in the definition of regular expressions.

- The **union** of two regular expressions, $A$ and $B$, is represented by arranging the corresponding digraphs in parallel leading from a new source to a new sink. All new edges are labeled $\varepsilon$.

- The **concatenation** of $A$ and $B$ is represented by arranging the corresponding digraphs in sequence, adding an edge labeled $\varepsilon$ between the sink of the graph for $A$ and the source of the graph for $B$.

- The **closure** of $A$ is represented by forming a loop created by adding an edge labeled $\varepsilon$ from the sink of $A$ to the source of $A$. In addition, we create a new source replacing and connecting it to the old source, and a new sink replacing and connecting it to the old sink, and an edge bypassing the entire old graph by connecting the new source directly to the new sink.

In order to construct the digraph of a regular expression, we would first turn the expression into Polish or postfix notation and then construct the graph by evaluation the expression in that notation. Note that the digraph of $A$ has at most $2m$ vertices. This is true in the two base cases and is maintained inductively for the union, concatenation, and closure operations. Note also that by construction each vertex has at most two outgoing edges. This implies that the number of edges is at most $4m$.

**Maximal Paths.** How does the digraph $G$ represent the corresponding regular expression $A$, and how do we use it to solve the pattern matching problem? We need some definitions to address these questions. A **maximal path** in $G$ leads from the source to the sink, following edges in the right direction. The **string** of a path $p$, denoted by $s(p)$, is the concatenation of the edge-labels along $p$. The **language** of $G$ is the set of strings defined by maximal paths, $L(G) = \{s(p) \mid p$ is a maximal path in $G\}$. The definitions are illustrated in Figure 113, which shows the digraph of the regular expression for real numbers given above. We can prove inductively that the language of the digraph is the string set of the regular expression, $L(G) = S(A)$. It follows that determining whether or not a text $T$ belongs to $S(A)$ is equivalent to determining whether or not there is a maximal path $p$ with $s(p) = T$.

![Diagram of directed graph](image2)

Figure 113: Directed graph representing the set of unsigned fixed-point real numbers in binary notation. The empty string labels are not shown.

We simulate the nondeterministic behavior of the finite state machine by following all feasible paths in the digraph, which are the paths whose strings are prefixes of $T$. To do this, we represent each feasible path by its endpoint, implicitly assuming that it starts at the source. The collection of feasible paths is represented by a (possibly smaller) collection of endpoints. To extend the feasible paths, we read the next letter in the text and update the collection of endpoints. We have $T \in L(G)$ if there is a maximal path in the collection of feasible paths after reading the entire text $T$. Equivalently, $T \in L(G) = S(A)$ if the sink belongs to the final collection of endpoints.

**Searching the Digraph.** In the implementation of $G$, we assume a boolean field to mark or unmark a vertex and a character field to label an edge. The digraph $G$ itself is then given by a pair of pointers, one to its source and the other to its sink. For a text $T$, we have $T \in S(A)$ if there is a maximal path $p$ with $s(p) = T$. We cancel empty strings, so if $x = a_1a_2\ldots a_n$ then the sequence of labels of the maximal path $p$ should be $\varepsilon*a_1\varepsilon*a_2\varepsilon*\ldots \varepsilon*a_n\varepsilon*$. To decide whether or not there is such a maximal path, we use two data structures, the directed graph $G$ that rep-
resents $A$ and a queue $Q$ that maintains a subset of the vertices. We read $T$ from left to right and search the graph at the same time. In particular, after reading a prefix, $Q$ stores all vertices that can be reached from the source by following a path whose string is equal to that prefix. We alternate between adding vertices that can be reached by edges labeled $\varepsilon$ and adding vertices that can be reached by the next character in $T$. Finally, $T$ is a string in $S(A)$ iff after processing $T$ the sink of the directed graph is reachable and therefore stored in the queue.

Let $\alpha$ and $\omega$ be the source and the sink of the directed graph $G$ of $A$. Let $\text{REPLACE\_QUEUE}(a)$ be the function that replaces the current contents of the queue by all vertices that can be reached by edges labeled $a$, and let $\text{COMPLETE\_QUEUE}$ be the function that adds all vertices that can be reached by edges labeled $\varepsilon$.

\begin{verbatim}
ENQUEUE(a); COMPLETE\_QUEUE;
for i = 1 to n do
    REPLACE\_QUEUE(a_i); COMPLETE\_QUEUE
endfor;
return [$\omega \in Q$].
\end{verbatim}

As an example, consider searching the directed graph in Figure 113 with $T = 101.101$. Figure 114 illustrates the search by labeling a vertex with the integer $i$ if that vertex is stored in the queue after the $i$-th iteration of the for-loop. Note that the sink of the digraph belongs to the collection of endpoints after reading 1, 2, 3, 5, and 7 characters of the text. These are the prefixes that are legal real numbers by themselves.

Completion and Replacement. We look at the function $\text{COMPLETE\_QUEUE}$ in more detail. Besides the standard $\text{DEQUEUE}$ and $\text{ENQUEUE}$ operations we need an iteration through all vertices stored in the queue $Q$.

\begin{verbatim}
void COMPLETE\_QUEUE
for all $u \in Q$ do mark $u$ endfor;
for all $u \in Q$ do
    let $v_1, v_2$ be successors of $u$ in $G$;
    if $v_1$ is unmarked and label($uv_1$) = $\varepsilon$ then
        mark $v_1$; ENQUEUE($v_1$)
    endif
    endfor
endfor;
forall $u \in Q$ do unmark $u$ endfor.
\end{verbatim}

The procedure for replacement is similar, except that we dequeue vertices as we iterate through the queue. We may do this either by remembering where the queue changes from old to new vertices or by using two queues and alternating their roles in the algorithm.

Analysis. The running time for function $\text{COMPLETE\_QUEUE}$ is at most $O(m)$ because the queue cannot contain more vertices than there are in the digraph. For the same reason, the running time of function $\text{REPLACE\_QUEUE}$ is at most $O(m)$. To complete the entire algorithm, we run function $\text{REPLACE\_QUEUE}$ $n$ times and function $\text{COMPLETE\_QUEUE}$ $n + 1$ times. The total running time is therefore in $O(mn)$.